Technology Consulting Company
Research, Development &
Global Standard

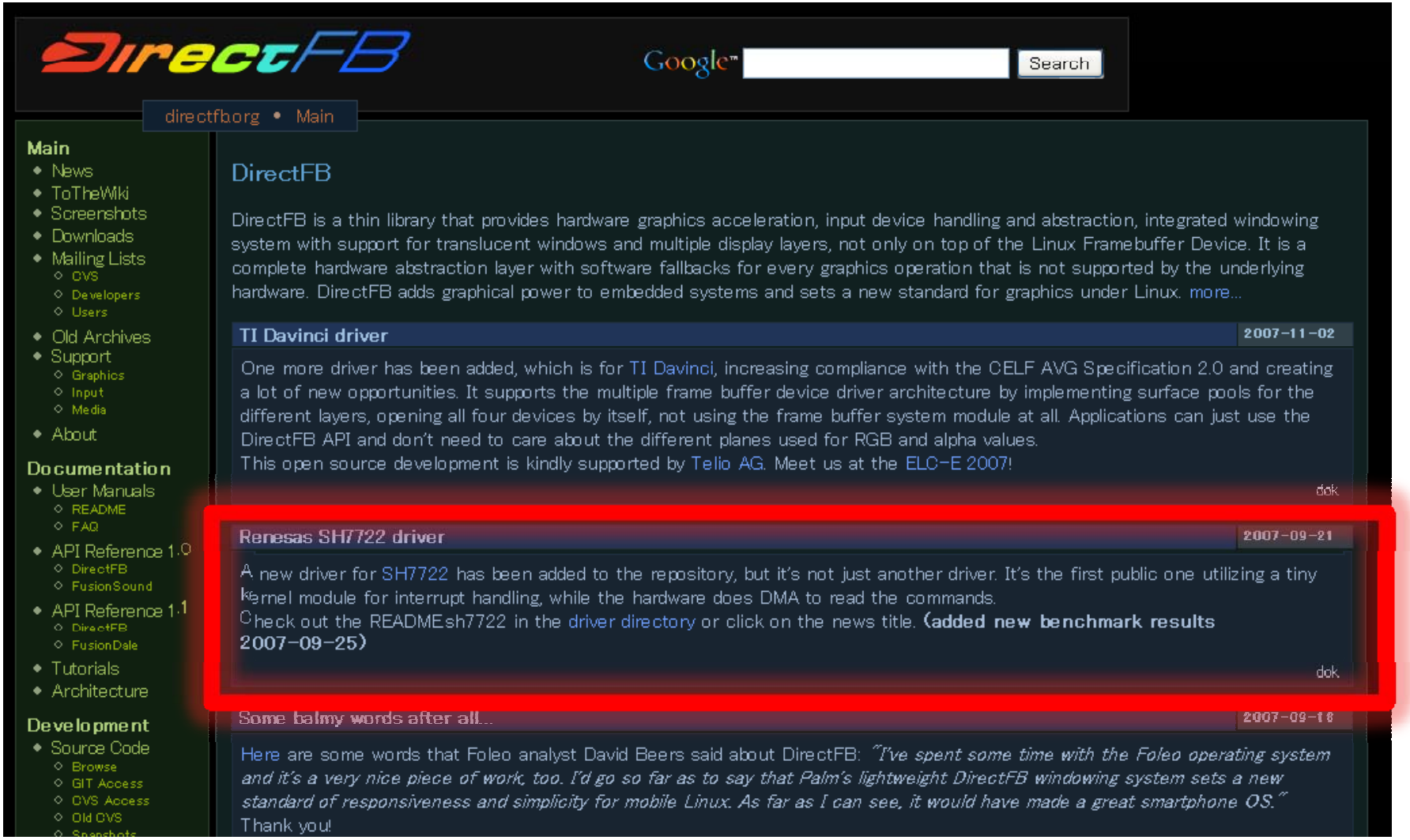
DirectFB Internals - Things You Need to Know to Write Your DirectFBgfxdriver

TakanariHayama, HisaoMunakata
and Denis Oliver Kropp

What is DirectFB?

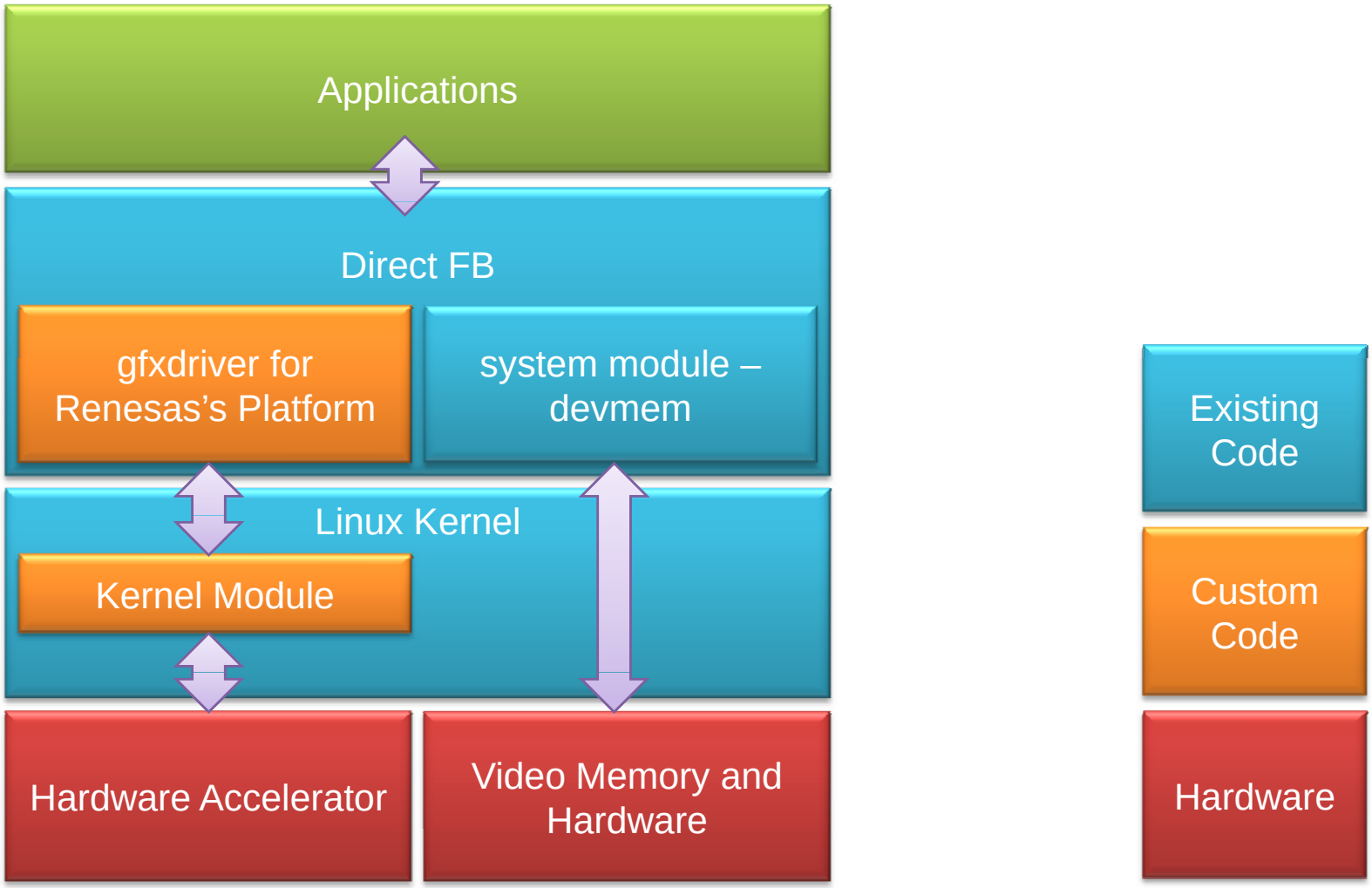
- Thin Graphics Library
  - Light weight and small footprint (< 700KB Library for SH4)
  - Not server/client model like X11
- Hardware Abstraction Layer for Hardware Graphics Acceleration
  - Anything not supported by Hardware still supported by software, i.e. utilize hardware where possible
- Multi-process support
- And others, e.g. build-in Window Manager etc.

The First Embedded Chip Support by the Mainline DirectFB – Renesas SH7722



df_andi and SawMan running on SH7722

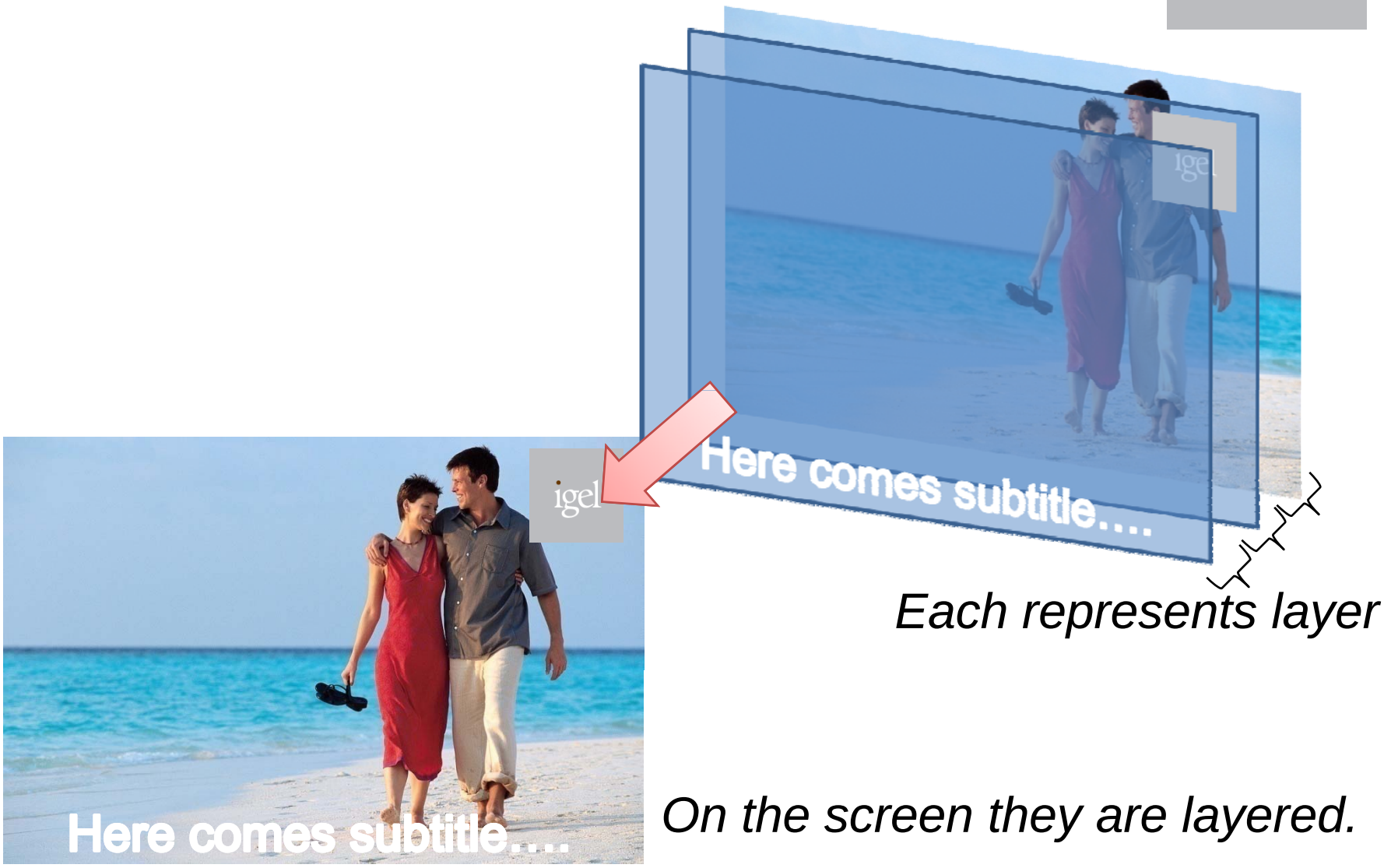
DirectFB Software Architecture for Renesas SH4 Platform



Important Terms in DirectFB

- Layers
  - Represents independent graphics buffers. Most of embedded devices have more than one layer. They get layered with appropriate alpha blending by hardware, and displayed.
- Surface
  - Reserved memory region to hold pixel data. Drawing and blitting operation in DirectFB is performed from/to surfaces. Memory of surfaces could be allocated from video memory or system memory depending on the given constraints.
- Primary Surface
  - Special surface that represents frame buffer of particular layer. If the primary surface is single buffered, any operation to this primary surface is directly visible to the screen.

Concept of Layers

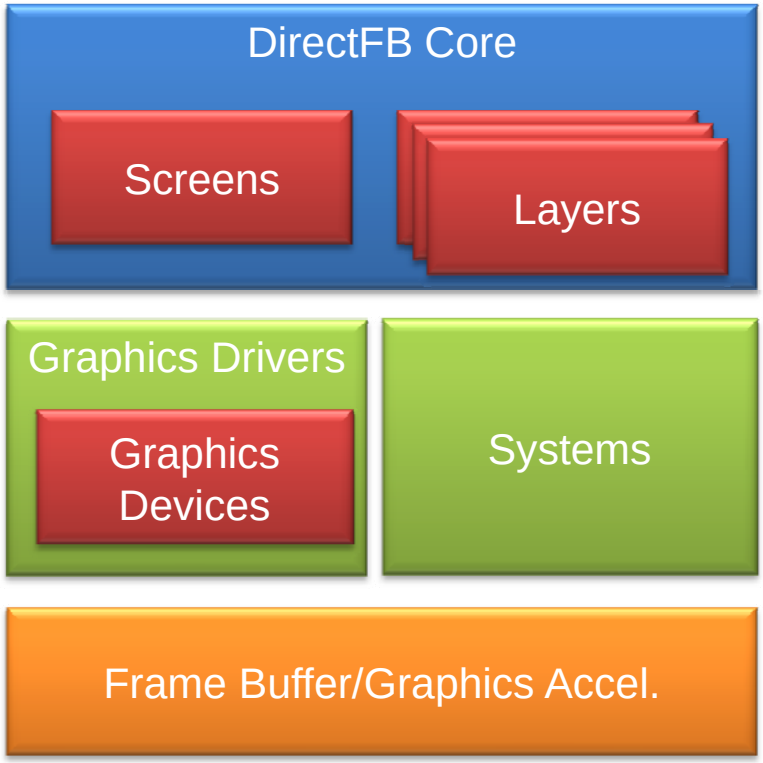


Each represents layer

On the screen they are layered.

DirectFB Internal Architecture

- Modules you need are:
  - Systems (optional)
    - `systems/*`
  - Graphics Devices
  - Graphics Drivers
    - `gfxdrivers/*`
  - Screens
  - Layers
- Each modules have defined function list that have to be implemented, i.e. DirectFB is designed more or less like object-oriented way.



Device Dependent Modules DirectFB

	Header	Module Declaration/Registration	Required Functions
Systems	`src/core/system.h` `src/core/core_system.h`	`DFB_CORE_SYSTEM(<name>)`	See `CoreSystemFuncs` in `core_system.h` and `system.h`
Graphics Drivers	`src/core/graphics_driver.h`	`DFB_GRAPHICS_DRIVER(<name>)`	See `GraphicsDriverFuncs` in `graphics_driver.h`
Graphics Devices	`src/core/gfxcard.h`	via `driver_init_driver()` in `GraphicsDriverFuncs`	See `GraphicsDeviceFuncs` in `gfxcard.h`
Screens	`src/core/screens.h`	`dfb_screens_register()`	See `ScreenFuncs` in `screens.h`
Layers	`src/core/layers.h`	`dfb_layers_register()`	See `DisplayLayerFuncs` in `layers.h`

Systems

- Frame buffer and hardware management.
  - Provides access to the hardware resources.
- Supported systems in DirectFB 1.1.0
  - fbdev (default)
  - osx
  - sdl
  - vnc
  - x11
  - devmem
- Can be specified in directfbrc

  e.g. `system=devmem`

For Embedded: system = devmem

- Merged in DirectFB 1.0.1. Uses /dev/mem to access to graphics hardware and frame buffer.
- Convenient way for those using memory mapped I/O and uniform memory among CPU and graphics accelerator.
  - Most of embedded devices fall into this.
- Additional parameters you must specify:
  - `video-phys=<hexaddress>`
    - `Physical start of video memory (devmem system)`
  - `video-length=<bytes>`
    - `Length of video memory (devmem system)`
  - `mmio-phys=<hexaddress>`
    - `Physical start of MMIO area (devmem system)`
  - `mmio-length=<bytes>`
    - `Length of MMIO area (devmem system)`
  - `accelerator=<id>`
    - `Accelerator ID selecting graphics driver (devmem system)`

How DirectFB matches systems and gfxdrivers?

- DirectFB core queries each gfxdrivers whether they support particular hardware accelerator by calling `driver_probe()` implemented in each gfxdrivers:
  - `intdriver_probe( CoreGraphicsDevice *device );`
- If the particular hardware is supported by the gfxdriver, the function shall return non-zero value, otherwise 0 must be returned.
- When devmem is used, the value passed to `driver_probe()` is the value passed by '`accelerator=<id>`'. Make sure the values match.

Graphics Drivers - gfxdriver

- Hardware specific graphics driver core. It consists of the following:
  - Graphics Driver Module
  - Graphics Device Module
  - Screen Module (optional for fbdev, but mandatory for devmem)
  - Layer Module (optional for fbdev, but mandatory for devmem)
- To have your graphics accelerator working, this is the code you must write at minimum!
  - You can use `devmem` for system. You don't have to write fbdev, if you don't feel like doing so.

Basics of Writing gfxdriver

1. There are bunch of headers you need to refer. Copy them from any gfxdrivers code you can find in `gfxdrivers/*` appropriately.
2. You must give unique name to the gfxdriver and declare using the following macro:
   - `DFB_GRAPHICS_DRIVER( sh7722 )`
3. The macro above requires 6 (six) functions to be defined for use by DirectFB core (see `src/core/graphics_drivers.h`). Define them.
   - Initialize/Close the gfxdriver
   - Initialize/Close the device
   - Retrieval of Metadata of the gfxdriver
4. Graphics acceleration capabilities are bound to the graphics devices via gfxdriver, you must appropriately specify what is supported. Set functions for the supported features. See `src/core/gfxcard.h` for details.

Functions you need to define in gfxdrivers

- `static int driver_probe( CoreGraphicsDevice *device );`
  - Should return non-zero value if the driver supports particular hardware passed by device. This is called by DirectFB core to probe which driver supports particular hardware in the system.
- `static void driver_get_info( CoreGraphicsDevice *device, GraphicsDriverInfo *info );`
  - Set the driver information in `info` and return.
- `static DFBResult driver_init_driver( CoreGraphicsDevice *device, GraphicsDeviceFuncs *funcs, void *driver_data, void *device_data, CoreDFB *core );`
  - Initialize the driver. After successively acquire all required resources, the driver should register screens and layers. Also needs to return list of callback functions for hardware accelerations via `funcs`.

Functions you need to define in gfxdrivers (contd.)


- ```
  static DFBResult
  driver_init_device( CoreGraphicsDevice *device,
  GraphicsDeviceInfo *device_info,
                      void               *driver_data,
                      void               *device_data );
  ```
  - Initialize hardware. All necessary hardware initialization should be processed here.
- ```
  static void
  driver_close_device( CoreGraphicsDevice *device,
                       void               *driver_data,
                       void               *device_data );
  ```
  - Whatever you need to do to while you close the device should come here.
- ```
  static void
  driver_close_driver( CoreGraphicsDevice *device,
                       void               *driver_data );
  ```
  - Whatever you need to do to while you close the driver should come here.
  ```

How gfxdriverGets Initialized

1. DirectFB calls `driver_probe()` in each gfxdriver on the system with a graphics device identifier to find appropriate gfxdriver for the device.
2. If `driver_probe()` of a gfxdriver returns non-zero, then the DirectFB calls `driver_init_driver()`. In `driver_init_driver()`:
   - Register graphics device functions
   - Register screen
   - Register layers
3. The DirectFB then calls `driver_init_device()` subsequently. In `driver_init_device()`:
   - Set capabilities supported by the device in `GraphicsDeviceInfo *device_info`, e.g. graphics acceleration capabilities such as Blit/Draw.

Graphics Device Functions

- You should set graphics device functions in `GraphicsDeviceFuncs *func` passed as an argument to `driver_init_driver()`.
- You don't have to set all functions. Set only those you really need.
- Most important ones are:
  - Reset/Sync graphics accelerator
  - Check/Set state of the graphics accelerator
  - Blitting/Drawing functions

22 Graphics Device Functions

- `void (*AfterSetVar)( void *driver_data, void *device_data );`
  - function that is called after variable screeninfo is changed (used for buggy fbdev drivers, that reinitialize something when calling FBIO_PUT_VSCREENINFO)
- `void (*EngineReset)( void *driver_data, void *device_data );`
  - Called after driver->InitDevice() and during dfb_gfxcard_unlock(true). The driver should do the one time initialization of the engine, e.g. writing some registers that are supposed to have a fixed value.
  - This happens after mode switching or after returning from OpenGL state (e.g. DRI driver).
- `DFBResult (*EngineSync)( void *driver_data, void *device_data );`
  - Makes sure that graphics hardware has finished all operations.
  - This method is called before the CPU accesses a surface' buffer that had been written to by the hardware after this method has been called the last time.
  - It's also called before entering the OpenGL state (e.g. DRI driver).

22 Graphics Device Functions (contd.)

- `void (*FlushTextureCache)( void *driver_data, void *device_data );`
  - after the video memory has been written to by the CPU (e.g. modification of a texture) make sure the accelerator won't use cached texture data
- `void (*FlushReadCache)( void *driver_data, void *device_data );`
  - After the video memory has been written to by the accelerator make sure the CPU won't read back cached data.
- `void (*SurfaceEnter)( void *driver_data, void *device_data, CoreSurfaceBuffer *buffer, DFBSurfaceLockFlags flags );`
  - Called before a software access to a video surface buffer.
- `void (*SurfaceLeave)( void *driver_data, void *device_data, CoreSurfaceBuffer *buffer );`
  - Called after a software access to a video surface buffer.

22 Graphics Device Functions (contd.)

- `void (*GetSerial)( void *driver_data, void *device_data, CoreGraphicsSerial *serial );`
  - Return the serial of the last (queued) operation.
  - The serial is used to wait for finishing a specific graphics operation instead of the whole engine being idle.
- `DFBResult (*WaitSerial)( void *driver_data, void *device_data, const CoreGraphicsSerial *serial );`
  - Makes sure that graphics hardware has finished the specified operation.
- `void (*EmitCommands) ( void *driver_data, void *device_data );`
  - emit any buffered commands, i.e. trigger processing.

22 Graphics Device Functions (contd.)

- `void (*InvalidateState)( void *driver_data, void *device_data );`
  - Called during dfb_gfxcard_lock() to notify the driver that the current rendering state is no longer valid.
- `void (*CheckState)( void *driver_data, void *device_data, CardState *state, DFBAccelerationMaskaccel );`
  - Check if the function 'accel' can be accelerated with the 'state'. If that's true, the function sets the 'accel' bit in 'state->accel'. Otherwise the function just returns, no need to clear the bit.
- `void (*SetState)  ( void *driver_data, void *device_data, struct _GraphicsDeviceFuncs *funcs, CardState *state, DFBAccelerationMaskaccel );`
  - Program card for execution of the function 'accel' with the 'state'. 'state->modified' contains information about changed entries. This function has to set at least 'accel' in 'state->set'. The driver should remember 'state->modified' and clear it. The driver may modify 'funcs' depending on 'state' settings.

22 Graphics Device Functions (contd.)

- `bool (*FillRectangle) ( void *driver_data, void *device_data, DFBRectangle *rect );`
- `bool (*DrawRectangle) ( void *driver_data, void *device_data, DFBRectangle *rect );`
- `bool (*DrawLine)      ( void *driver_data, void *device_data, DFBRegion *line );`
- `bool (*FillTriangle)  ( void *driver_data, void *device_data, DFBTriangle *tri );`
  - Drawing functions.

22 Graphics Device Functions (contd.)

- `bool (*Blit)  ( void *driver_data, void *device_data, DFBRectangle *rect, intdx, intdy );`
- `bool (*StretchBlit)  ( void *driver_data, void *device_data, DFBRectangle *srect, DFBRectangle *drect );`
- `bool (*TextureTriangles)( void *driver_data, void *device_data, DFBVertex *vertices, int num, DFBTriangleFormation formation );`
  - Blitting functions.

22 Graphics Device Functions (contd.)

- `void (*StartDrawing)( void *driver_data, void *device_data, CardState *state );`
  - Signal beginning of a sequence of operations using this state. Any number of states can be 'drawing'.
- `void (*StopDrawing)( void *driver_data, void *device_data, CardState *state );`
  - Signal end of sequence, i.e. destination surface is consistent again.

How DirectFB Calls Hardware Accelerator?

1. Check whether the particular functionality is supported by the hardware by calling `CheckState()` in the gfxdriver.
2. If the `CheckState()` tells DirectFB that particular function is supported by the hardware, then DirectFB subsequently calls `SetState()`. Otherwise, it falls back to the software rendering.
3. In the `SetState()`, all required parameters shall be taken and prepared to be passed to the hardware.
4. After it returns from `SetState()`, DirectFB finally calls the appropriate drawing/blitting functions, e.g. `Blit()`.

Advanced Feature – Queuing Draw/Blit Commands

- Some graphics accelerator supports command queuing or command lists. To utilize this feature, you may queue draw/blit as much as you can, and then kick the hardware.
- To do this, `EmitCommands()` should be defined. See the example in sh7722 gfxdriver.

Screens

- Screens represent output device, e.g. LCD.
- If you have fixed size screen, the minimal functions you need to define are:
  - `InitScreen()`
  - `GetScreenSize()`
- For more details, see `src/core/screens.h`.

Layers

- Layers represent independent graphics buffers.
- They’re basically converged by hardware when they get displayed on the screen.
  - Normally, alpha blending is applied.
- Layers are required to be able to:
  - Change size, pixel format, buffering mode and CLUT.
  - Flip buffer.
- For more details, see `src/core/layers.h`.

Important DisplayLayerFuncs

- LayerDataSize()
  - Returns size of layer data to be stored in shared memory.
- RegionDataSize()
  - Returns size of region data to be stored in shared memory.
- InitLayer()
  - Initialize layer. Called only once by master process.
- TestRegion()
  - Check if given parameters are supported.
- SetRegion()
  - Program hardware with given parameters.
- RemoveRegion()
  - Remove the region.
- FlipRegion()
  - Flip the frame buffer.

Layers – Change Configuration

1. The DirectFB core calls `TestRegion()` first, to see if the configuration is supported or not. `TestRegion()` should return `DFB_OK`, if the configurations are supported. Otherwise, `DFB_UNSUPPORTED` and details should be returned.
2. The DirectFB core then calls `SetRegion()`. In the `SetRegion()`, you should apply all changes to the hardware.

Layers – Flip

- If you support double buffer or triple buffer, you should implement the feature in `FlipRegion()`.
- `FlipRegion()` is called whenever the flip is needed.

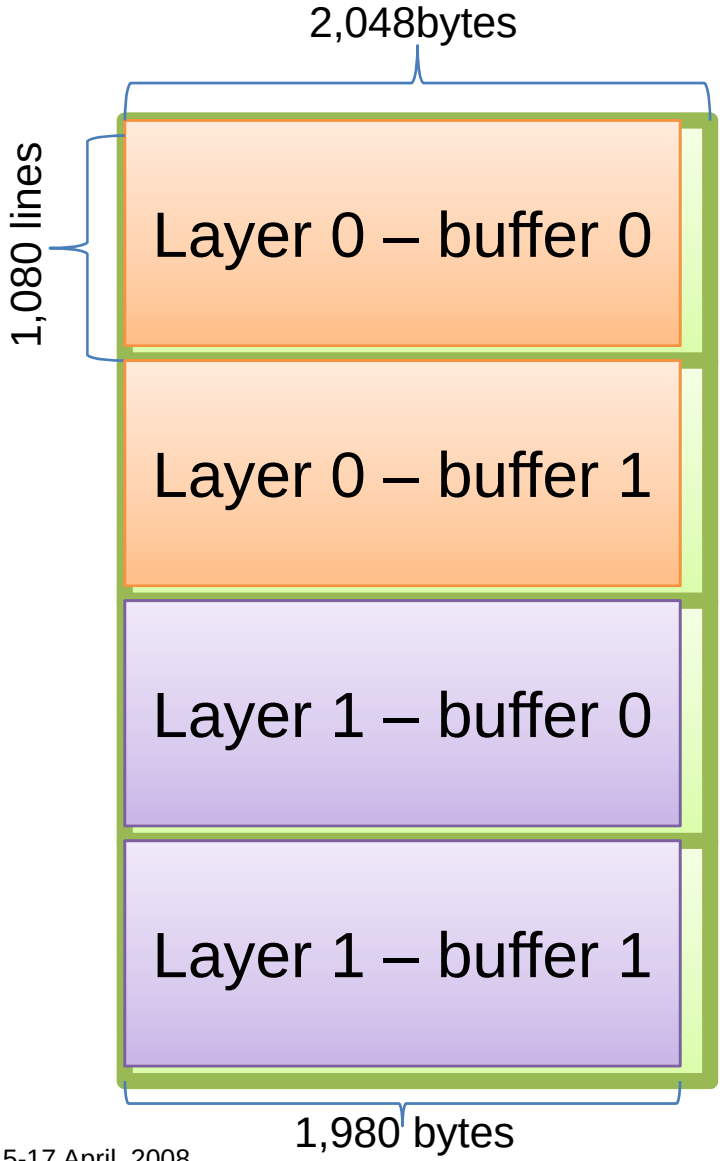
Surface Allocation

- DirectFB 1.0 used use single one-dimensional linear Surface allocator.
  - When a surface is requested with DSCAPS_VIDEOONLY, built-in surface manager allocated surface from a contiguous memory block.
  - When a surface is requested with DSCAPS_SYSTEMONLY, the surface is allocated using malloc().
  - For embedded graphics accelerator, e.g. to utilize simple blitter, you're likely required to use physically contiguous memory, i.e. DSCAPS_VIDEOONLY is only option.
- Only way to customize surface allocation was through Layer Driver API, mostly for primary surfaces.
  - Rarely used in R1, although quite interesting feature.

Allocating Primary Surface Your Way

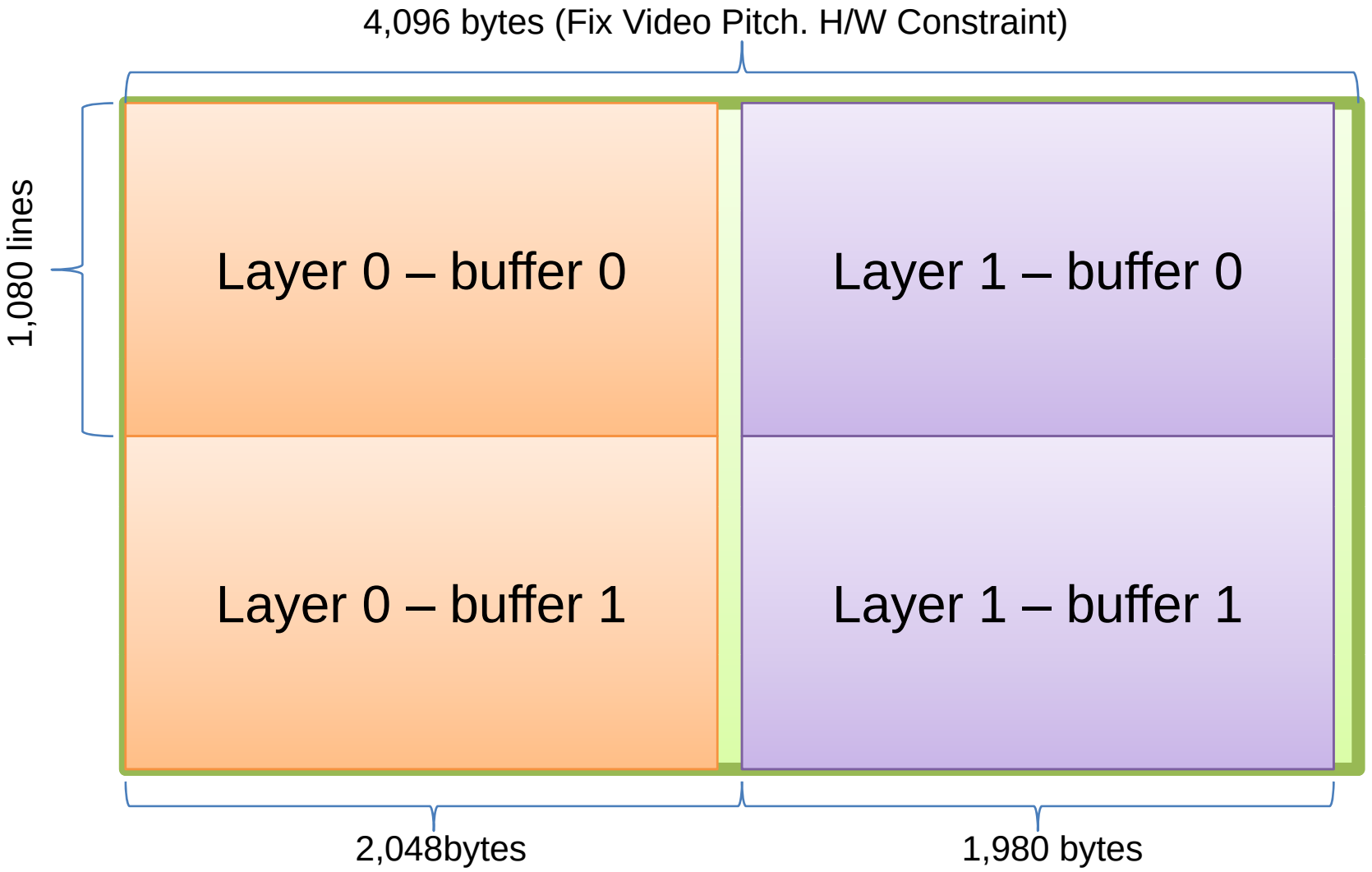
- DirectFB automatically allocates primary surface by using given size, pixel format, byte pitch alignment and byte offset alignment values.
  - For most hardware, this is enough.
- However, some hardware requires specific way to allocate frame buffer.
  - E.g., a hardware requires 4KB fixed size video pitch, and for memory efficiency it requires two layers to be allocated side-by-side.
- You can always override surface allocation mechanism, and allocates memory your way.
  - This is done in fbdev system as well.

Allocating Primary Surface Your Way (Contd.) – Normal Memory Allocation



This is typical frame buffer allocation for 2 double buffered layers with CLUT8 in Full HD, i.e. 1980x1080x8bit.

Allocating Primary Surface Your Way (Contd.) – Optimized Allocation



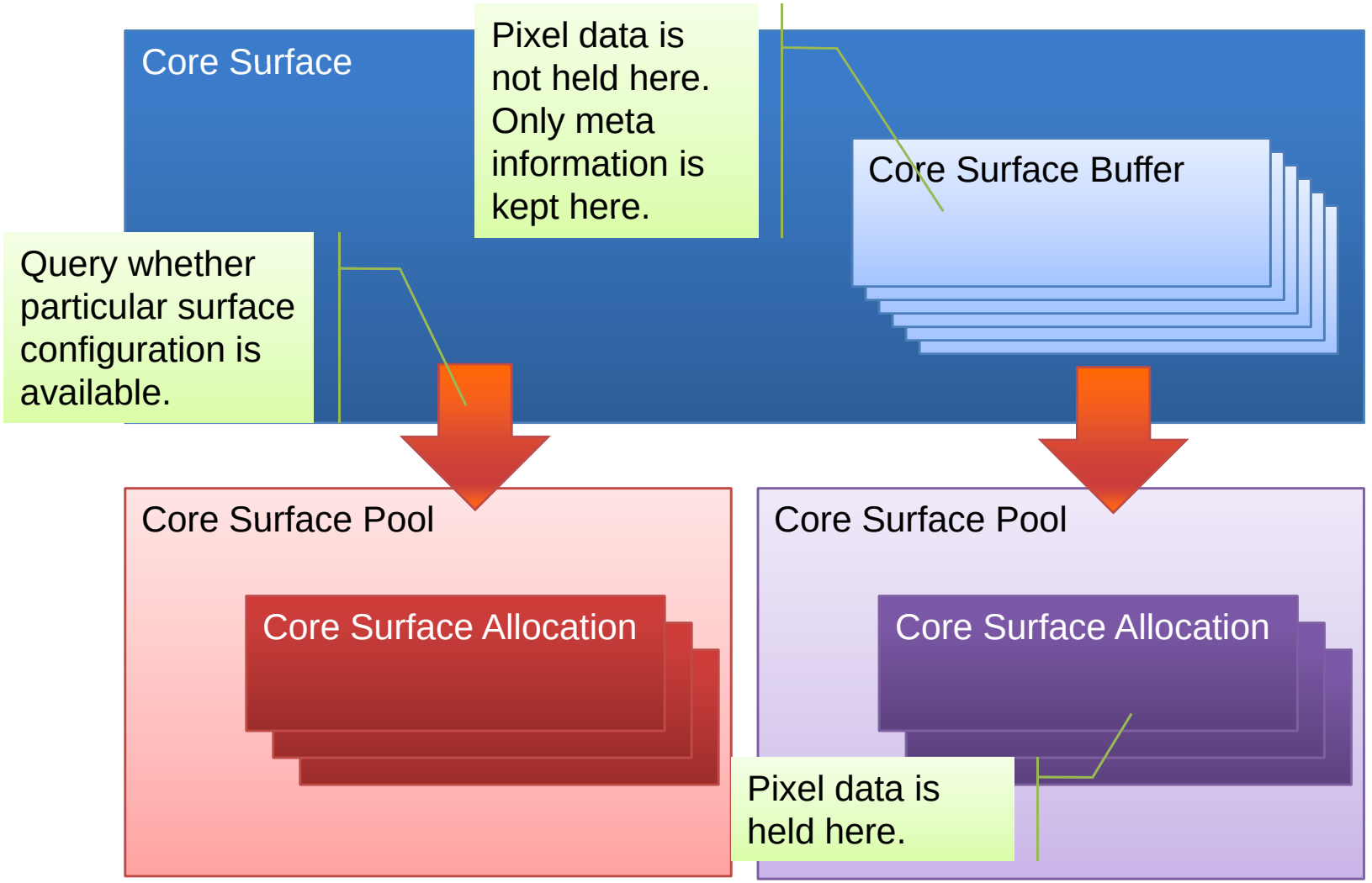
Allocating Primary Surface Your Way (Contd.)

1. Define `AllocateSurface()` and `ReallocateSurface()` in DisplayLayerFuncs.
   - You should just make sure you create an appropriate container for surface, i.e. `CoreSurface`.
2. Allocate or reallocate surface in `SetRegion()`.
   - May claim video memory by calling `dfb_surface_create()`, and resize if necessary with `dfb_surface_reformat()`.
   - Should claim the memory from video memory.
   - Don't forget to call `dfb_surface_globalize()` to make this surface visible to others as well.
   - Set primary surface's video memory offset appropriately: `front_buffer->video.offset`, `back_buffer->vidoe.offset`, and `idle_buffer->video.offset`.

New Surface Pool

- Starting from DirectFB 1.1, DirectFB introduced new concept – Surface Pool.
  - http://www.directfb.eu/wiki/index.php/DirectFB_2.0:_Surface_Pools
- Now you can manage surface buffer your way!

New Surface Pool (contd.)



New Surface Pool (contd.)

- Surface Pools are the new key abstraction to break out of this limited model, which only did a good job for Linux' Frame Buffer Device or other Low Level APIs, also (part of) /dev/mem.
- The big step was moving away from the hardcoded system<->video heap logic including sync, transfer, locking, etc. to a generic mechanism with any number of heaps/pools with totally different capabilities, allowing system/driver module to provide their own implementations (pools) just like input drivers (devices).
- Surface Pool Negotiation is one of the key aspects, because it routes allocation requests for surface buffers to the correct pool, or the best suitable when more than one is supporting all required access flags, e.g. CPU/GPU, surface type flags, e.g. Layer or Font, and other criteria that will be added.
- After a surface has been created as a CoreSurface (FusionObject), based on the chosen CoreSurfaceConfiguration, the CoreSurfaceBuffer structures are allocated and added without any actual allocation of pixel data happening. Even after returning, the buffers are still virtual entities. The first Lock or Write will trigger the negotiation, before allocation/locking in the right pool happens.

From:[*http://www.directfb.eu/wiki/index.php/DirectFB_2.0:_Surface_Pools*](http://www.directfb.eu/wiki/index.php/DirectFB_2.0:_Surface_Pools)

HAVE FUN WITH DIRECTFB!